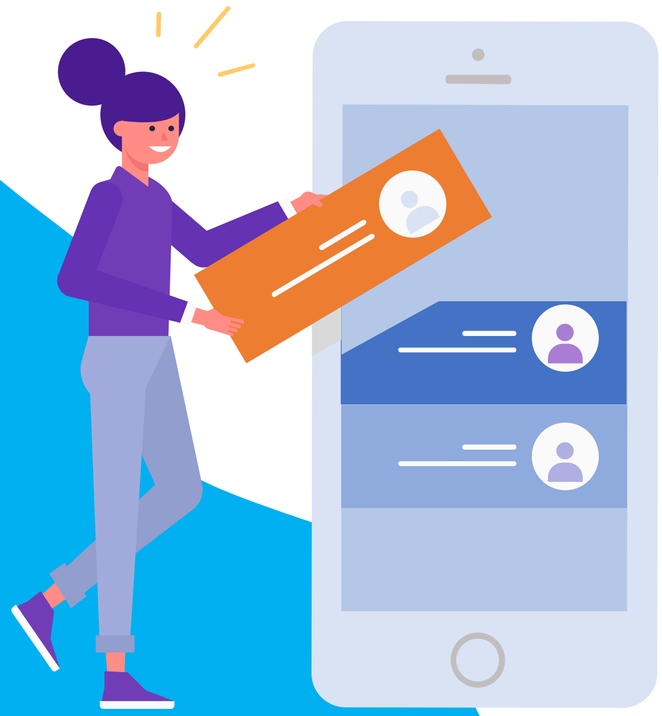
MOJADIPRO

FLUTTER STATE MANAGEMENT

MOJADIPRO

MATERI



01 Perbedaan Antara State & State Management

02 List View, List View Builder, dan Bottom Sheet Widget

03 Architecture Design Pattern

04 Provider

MOJADIPRO

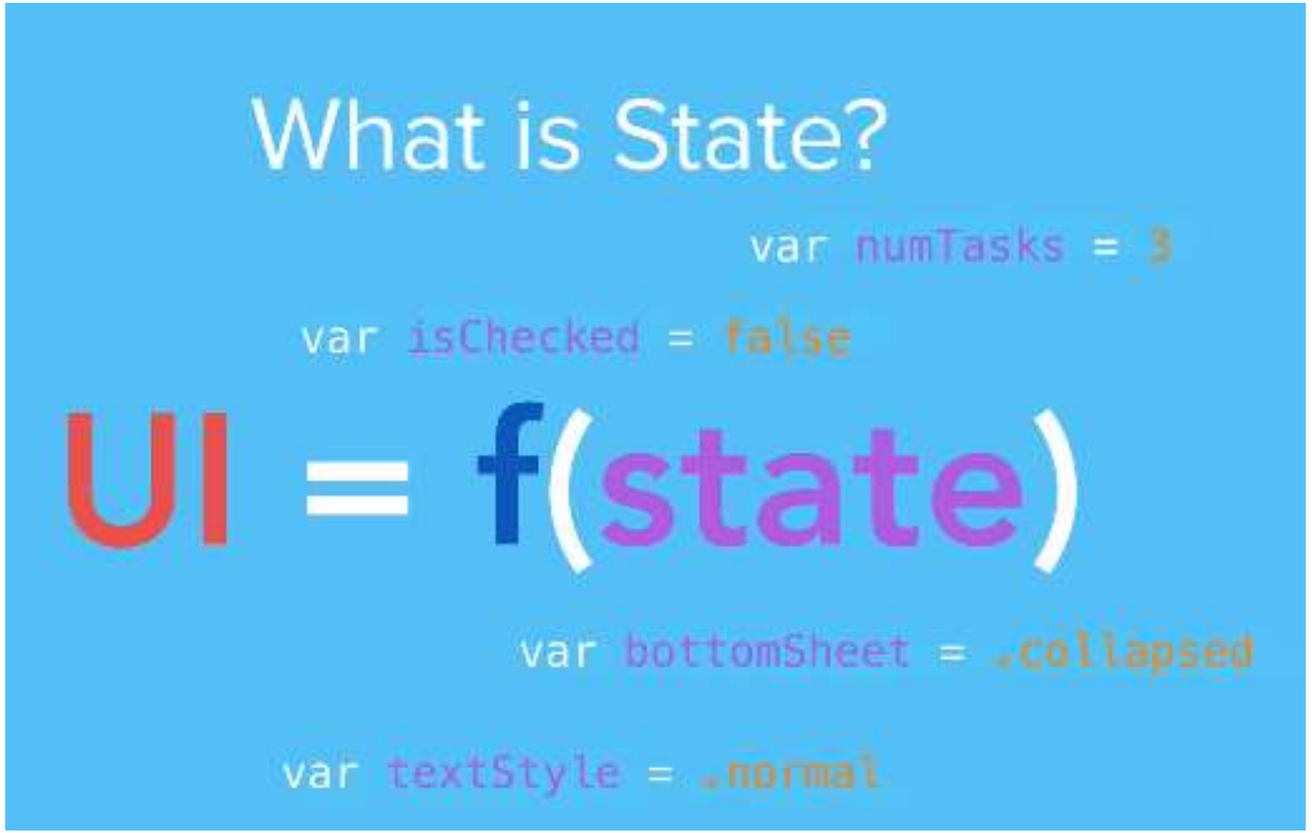
01

Perbedaan Antara State & State Management

MOJADIPRO

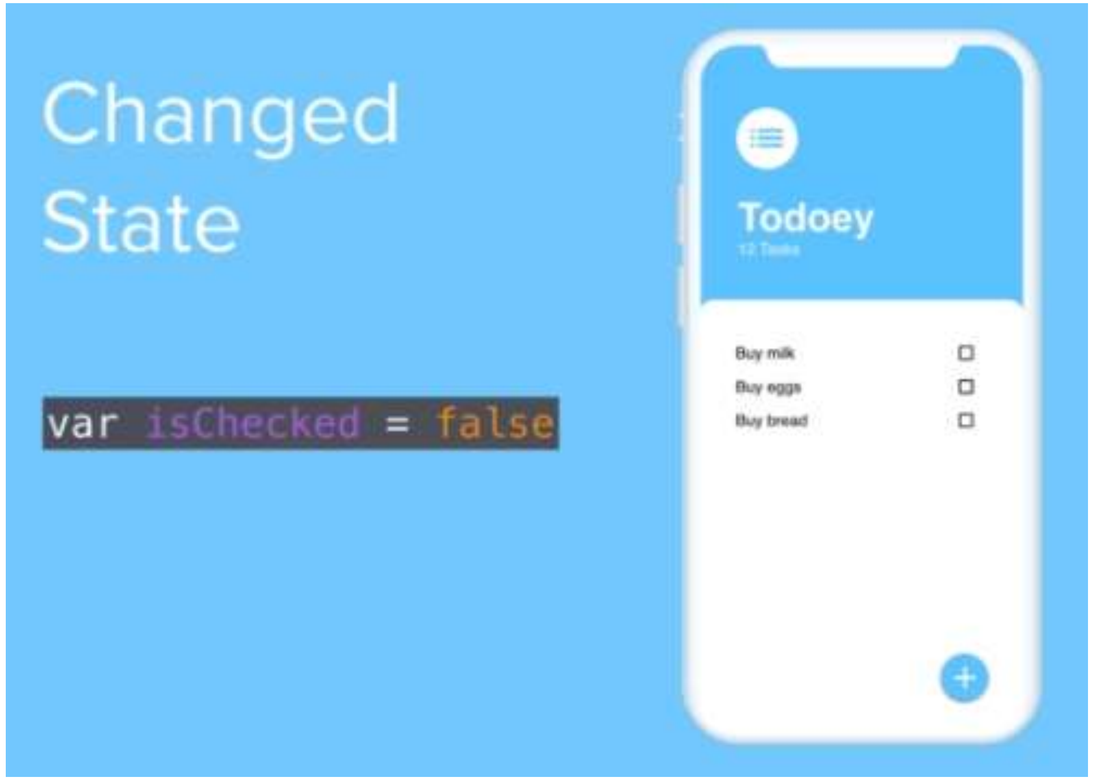
State

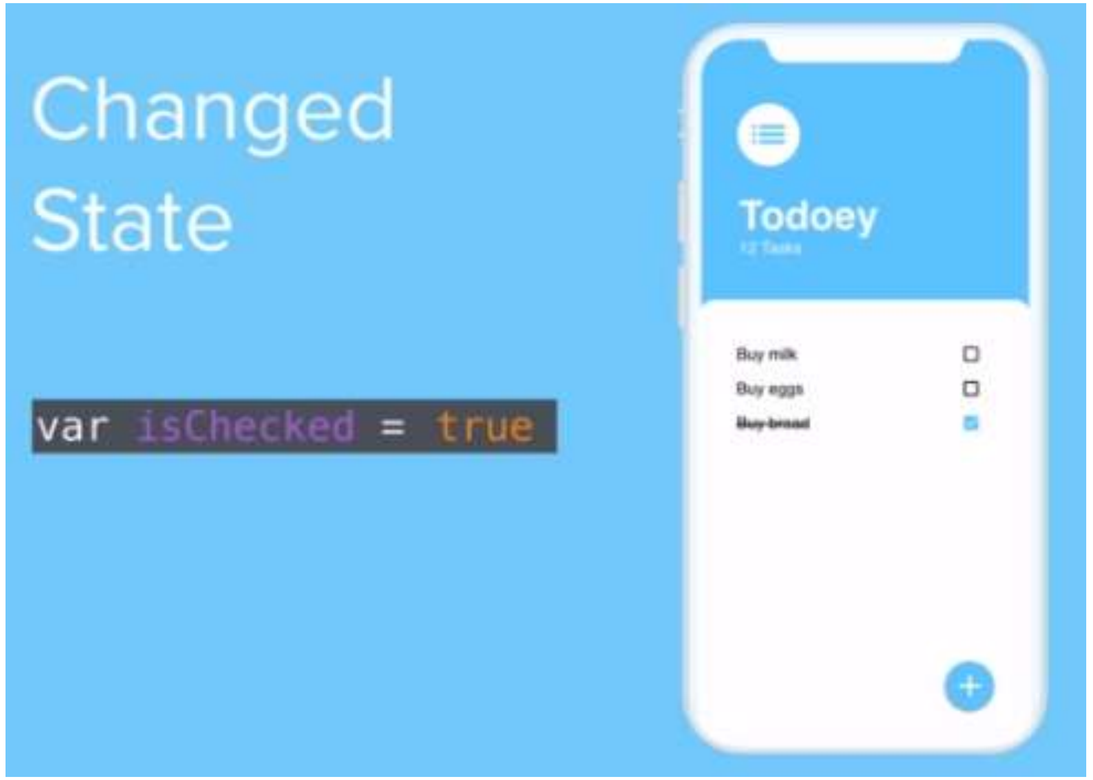
State dapat merujuk pada data yang ada dalam aplikasi. State adalah nilai dari semua variabel yang bersama-sama membentuk antarmuka pengguna.



MOJADIPRO

State





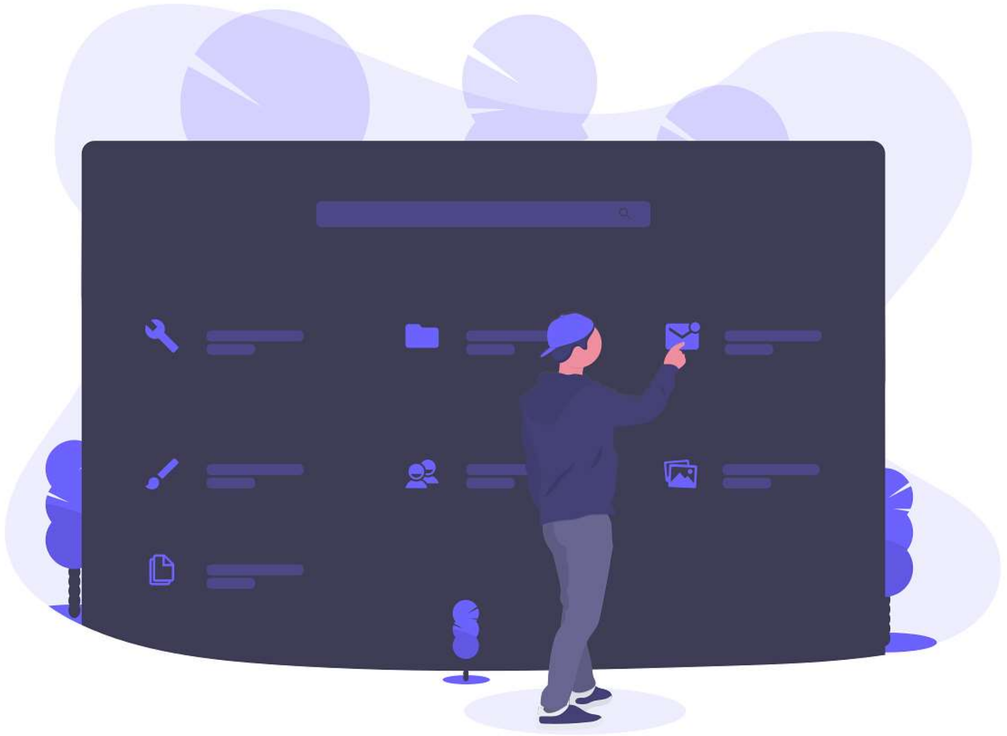
Terdapat variabel bernama isChecked dan kita akan menyetelnya ke false secara default.
Jika isChecked sama dengan false, maka semua kotak centang kosong. Tetapi ketika pengguna mengetuk salah satu kotak centang dan kita mengubah nilai variabel itu menjadi true, maka itu akan memperbarui antarmuka pengguna

MOJADIPRO

State Management

State management berkaitan dengan bagaimana aplikasi mengelola dan memperbarui state serta menyediakan cara untuk berbagi dan mengakses state di berbagai bagian aplikasi yang berbeda.

State management dapat mencakup penggunaan pola desain, library, atau framework khusus yang memfasilitasi pengelolaan state.



MOJADIPRO

02

List View, List View Builder, Dan Bottom Sheet Widget

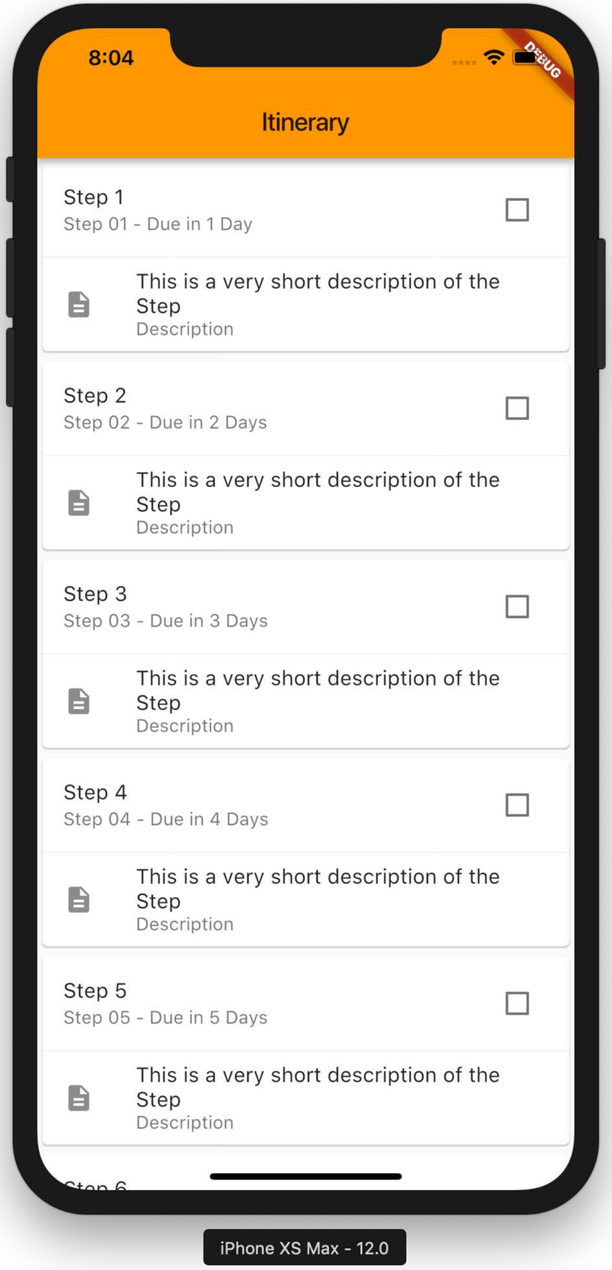
MOJADIPRO

List View

ListView adalah sebuah widget yang digunakan untuk menampilkan daftar item yang dapat di-scroll secara vertikal

Contoh penggunaan ListView:

```
ListView(
  children: <Widget>[
    ListTile(
      title: Text('Item 1'),
    ),
    ListTile(
      title: Text('Item 2'),
    ),
    ListTile(
      title: Text('Item 3'),
    ),
    // ...
  ],
)
```



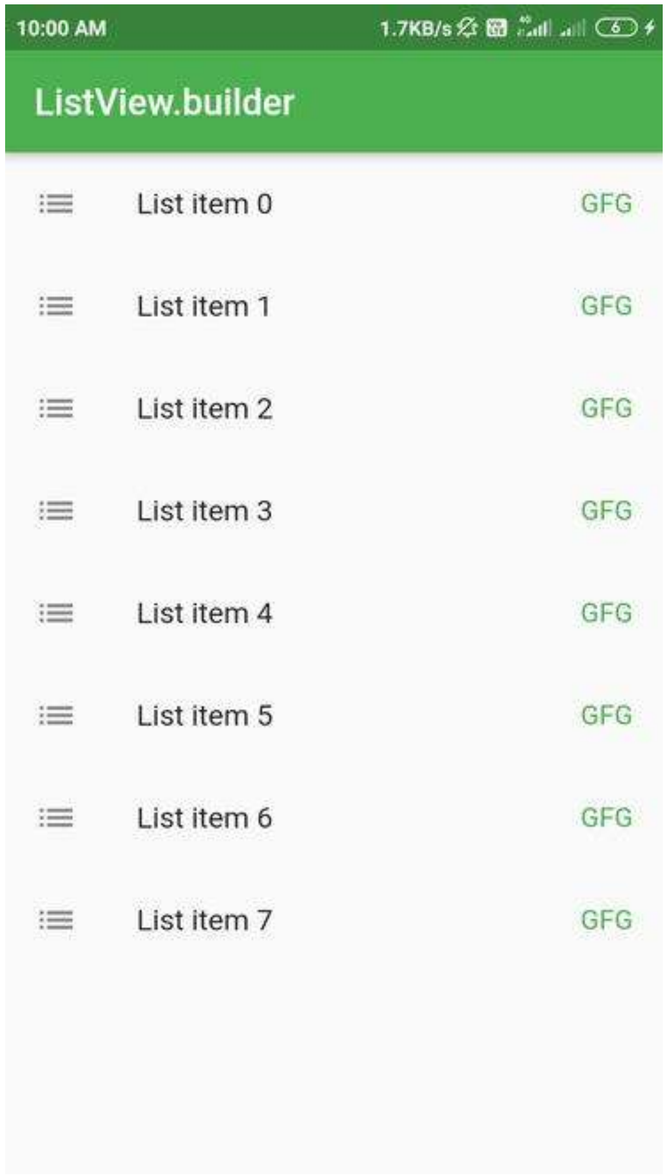
MOJADIPRO

List View Builder

ListView.builder adalah jenis ListView yang lebih efisien untuk menampilkan daftar item yang berjumlah besar atau dinamis. Dalam ListView ini, item hanya dibangun saat diperlukan saat pengguna menggulir layar, sehingga meminimalkan penggunaan memori dan meningkatkan kinerja.

Contoh penggunaan ListViewBuilder:

```
ListView.builder(
  itemCount: myItemList.length,
  itemBuilder: (BuildContext context, int index) {
    return ListTile(
      title: Text(myItemList[index]),
    );
  },
)
```



MOJADIPRO

Widget Bottom Sheet

Widget BottomSheet menyediakan panel geser yang ditampilkan di bagian bawah layar. BottomSheet biasanya digunakan untuk menyajikan informasi, tindakan, atau pengaturan tambahan tanpa sepenuhnya menghalangi konten yang mendasarinya.

Widget BottomSheet dapat memiliki status berbeda, seperti diperluas, diciutkan, atau disembunyikan, bergantung pada interaksi pengguna.

MOJADIPRO

Contoh Widget Bottom Sheet

```
import 'package:flutter/material.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'BottomSheet Example',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(),
    );
  }
}
```

```
class MyHomePage extends StatelessWidget {
  void _showBottomSheet(BuildContext context) {
    showModalBottomSheet(
      context: context,
      builder: (BuildContext context) {
        return Container(
          height: 200,
          child: Center(
            child: Text(
              'This is the bottom sheet',
              style: TextStyle(fontSize: 20),
            ),
          ),
        );
      },
    );
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('BottomSheet Example'),
      ),
      body: Center(
        child: RaisedButton(
          onPressed: () {
            _showBottomSheet(context);
          },
          child: Text('Show BottomSheet'),
        ),
      ),
    );
  }
}
```

MOJADIPRO

03

Architecture Design Pattern

MOJADIPRO

Architecture Design Pattern

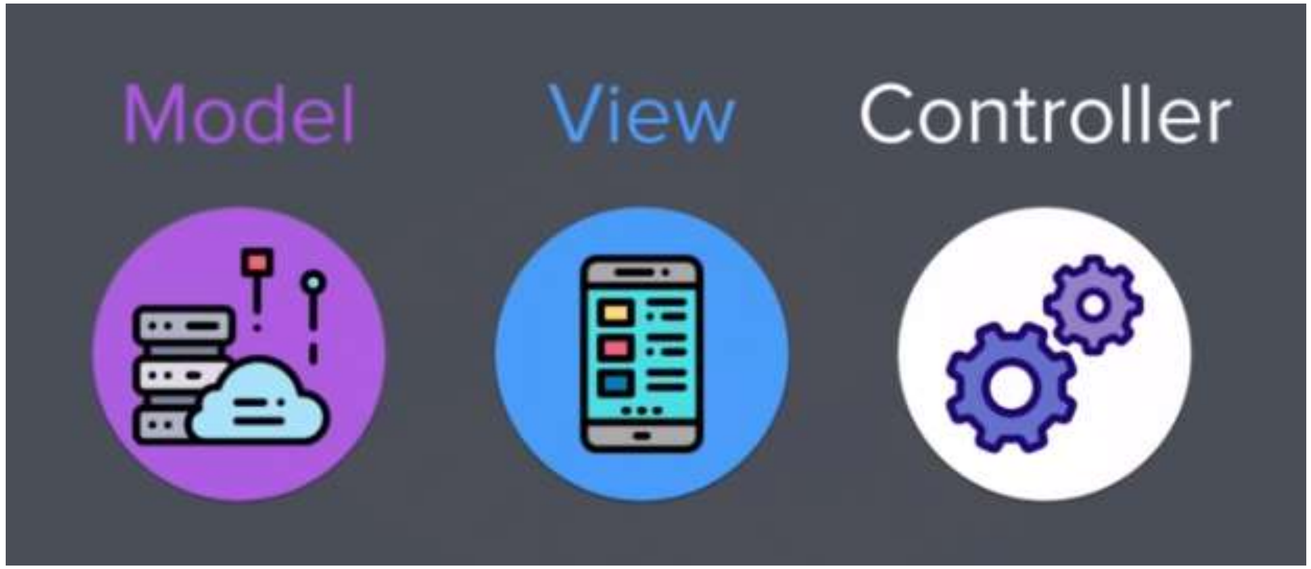
Architecture design pattern dalam aplikasi Flutter adalah kerangka kerja yang membantu mengatur struktur, logika, dan aliran data dalam aplikasi.
Desain pola ini membantu memisahkan tugas-tugas yang berbeda, meningkatkan pengujian, dan memudahkan pemeliharaan kode.

Desain pola arsitektur umum yang sering digunakan dalam aplikasi Flutter:

1. Model-View-Controller (MVC):
2. Model-View-ViewModel (MVVM):
3. BLoC (Business Logic Component):
4. Provider.
5. Redux.
6. Clean Architecture:

MVC

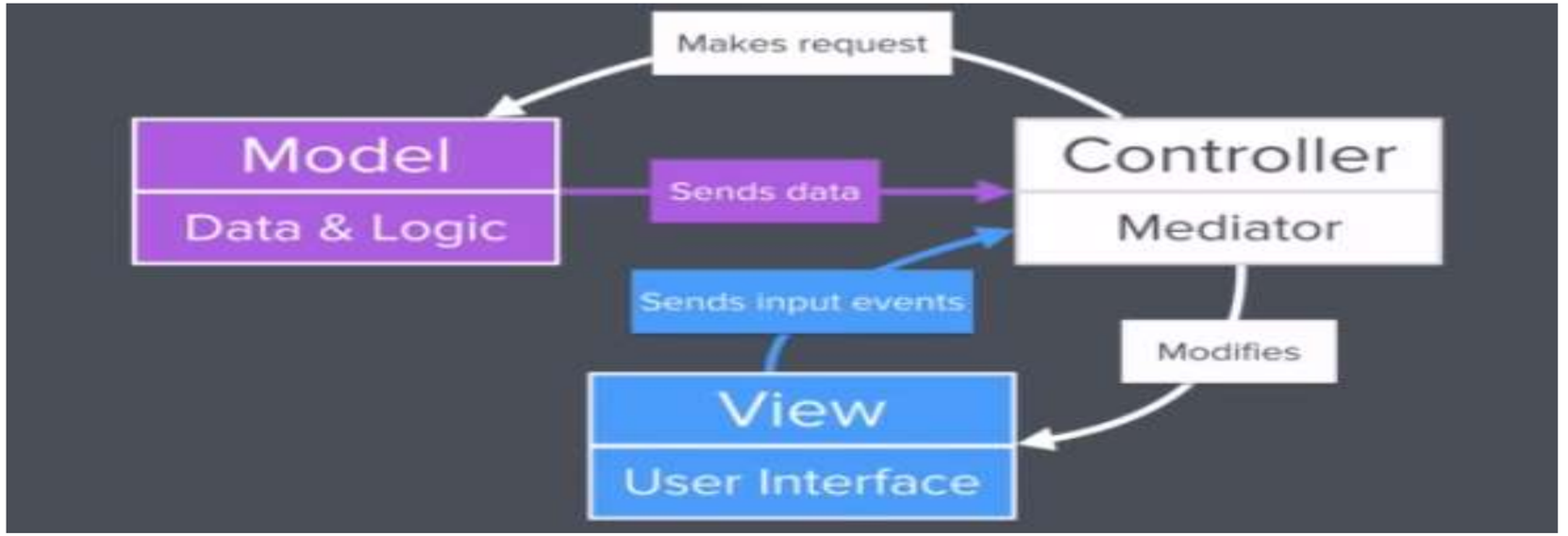
Salah satu arsitektur paling populer untuk seluler adalah Model View Controller.
Dan ini pertama kali dipopulerkan oleh Apple dalam mengembangkan aplikasi iOS.



1. Model = mengelola data, menangani data dan logika.
2. View/tampilan = mengelola apa yang masuk ke layar, menangani antarmuka pengguna
3. Controller/pengontrol = perantara antara semua komponen ini, menjadi mediator

MOJADIPRO

Cara Kerja MVC



1. Ketika pengguna mengetuk sesuatu di aplikasi, kemudian kode dalam tampilan akan mengirim pesan tentang kejadian input tersebut ke pengontrol.
2. Controller kemudian akan menggunakan kejadian input tersebut untuk memutuskan apa yang harus dilakukan selanjutnya, dan itu akan membuat permintaan ke model untuk meminta beberapa data.
3. Sekarang kelas model berhubungan dengan database atau penyimpanan data dan akhirnya mengirimkan kembali data ke controller.
4. Pengontrol kemudian menggunakan data tersebut untuk mengubah tampilan.

MOJADIPRO

Pemrograman Imperative

Pemrograman imperative adalah pendekatan yang lebih tradisional dalam pengembangan perangkat lunak. Dalam pemrograman imperative, Anda secara eksplisit mendefinisikan langkah-langkah yang harus dijalankan oleh komputer untuk mencapai hasil yang diinginkan. Dalam konteks Flutter, ini berarti Anda secara langsung mengontrol alur eksekusi kode dan mengubah state secara langsung.

Dalam pemrograman imperative Flutter, Anda mengubah state secara langsung ketika ada peristiwa atau tindakan yang mempengaruhi UI. Misalnya, ketika tombol ditekan, Anda akan mengubah state dan mengupdate UI secara langsung. Anda akan menentukan bagaimana UI berubah berdasarkan perubahan state ini.

MOJADIPRO

Contoh Pemrograman Imperative

```
class Counter extends StatefulWidget {
  @override
  _CounterState createState() => _CounterState();
}

class _CounterState extends State<Counter> {
  int _count = 0;

  void _increment() {
    setState(() {
      _count++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Counter')),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text('Count: $_count'),
            RaisedButton(
              child: Text('Increment'),
              onPressed: _increment,
            ),
          ],
        ),
      ),
    );
  }
}
```

MOJADIPRO

Pemrograman Reactive

Pemrograman reactive, di sisi lain, adalah pendekatan yang berfokus pada aliran data dan reaksi terhadap perubahan. Dalam pemrograman reactive Flutter, Anda mendefinisikan hubungan antara data dan UI, dan perubahan data secara otomatis memicu pembaruan pada UI.

Dalam pemrograman reactive Flutter, Anda menggunakan konsep-konsep seperti Stream dan StreamBuilder untuk membuat aliran data yang memancar (stream) dan mengganti UI secara otomatis ketika ada perubahan pada aliran data tersebut.

MOJADIPRO

Contoh Pemrograman Reactive

```
class Counter {
  Stream<int> countStream = Stream<int>.periodic(Duration(seconds: 1), (count) => count + 1);
}

class CounterWidget extends StatelessWidget {
  final Counter counter = Counter();

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Counter')),
      body: Center(
        child: StreamBuilder<int>(
          stream: counter.countStream,
          builder: (context, snapshot) {
            if (snapshot.hasData) {
              return Text('Count: ${snapshot.data}');
            } else {
              return Text('Loading...');
            }
          },
        ),
      ),
    );
  }
}
```

MOJADIPRO

Prop Drilling

Prop drilling adalah proses mengirimkan data melalui beberapa lapisan widget dalam pohon widget Flutter. Ketika data diperlukan di dalam widget yang berada di lapisan terdalam, data tersebut harus dilewatkan melalui semua widget di antara widget tersebut secara eksplisit. Ini bisa menjadi tidak efisien dan membingungkan ketika aplikasi menjadi kompleks dengan banyak widget.

Contoh Prop Drilling: Misalkan kita memiliki aplikasi Flutter sederhana yang terdiri dari tiga widget yaitu WidgetA, WidgetB, dan WidgetC. WidgetA adalah widget terluar yang memiliki data yang akan digunakan oleh WidgetC yang berada di dalamnya.

MOJADIPRO

Contoh Prop Drilling

```
class WidgetA extends StatelessWidget {
  final String data;

  WidgetA(this.data);

  @override
  Widget build(BuildContext context) {
    return WidgetB(data);
  }
}

class WidgetB extends StatelessWidget {
  final String data;

  WidgetB(this.data);

  @override
  Widget build(BuildContext context) {
    return WidgetC(data);
  }
}

class WidgetC extends StatelessWidget {
  final String data;

  WidgetC(this.data);

  @override
  Widget build(BuildContext context) {
    return Text(data);
  }
}
```

MOJADIPRO

Lifting State Up

Lifting state up adalah teknik untuk mengatasi masalah prop drilling dengan mengangkat (mengangkat) state ke widget yang berada di tingkat yang lebih tinggi dalam pohon widget. Dengan mengangkat state, kita dapat membagikan data ke beberapa widget yang membutuhkannya tanpa harus melewati semua widget di antara mereka.

Contoh Lifting State Up: Misalkan kita memiliki aplikasi Flutter yang memungkinkan pengguna untuk mengklik tombol dan mengubah teks yang ditampilkan pada layar. Kita memiliki dua widget, yaitu ParentWidget dan ChildWidget. Saat tombol diklik di ChildWidget, teks yang ditampilkan harus berubah di ParentWidget.

MOJADIPRO

Contoh Lifting State Up

```
class ParentWidget extends StatefulWidget {
  @override
  _ParentWidgetState createState() => _ParentWidgetState();
}

class _ParentWidgetState extends State<ParentWidget> {
  String text = 'Hello';

  void changeText(String newText) {
    setState(() {
      text = newText;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Text(text),
        ChildWidget(changeText),
      ],
    );
  }
}

class ChildWidget extends StatelessWidget {
  final Function(String) callback;

  ChildWidget(this.callback);

  @override
  Widget build(BuildContext context) {
    return RaisedButton(
      onPressed: () {
        callback('World');
      },
      child: Text('Change Text'),
    );
  }
}
```

MOJADIPRO

04

Provider

MOJADIPRO

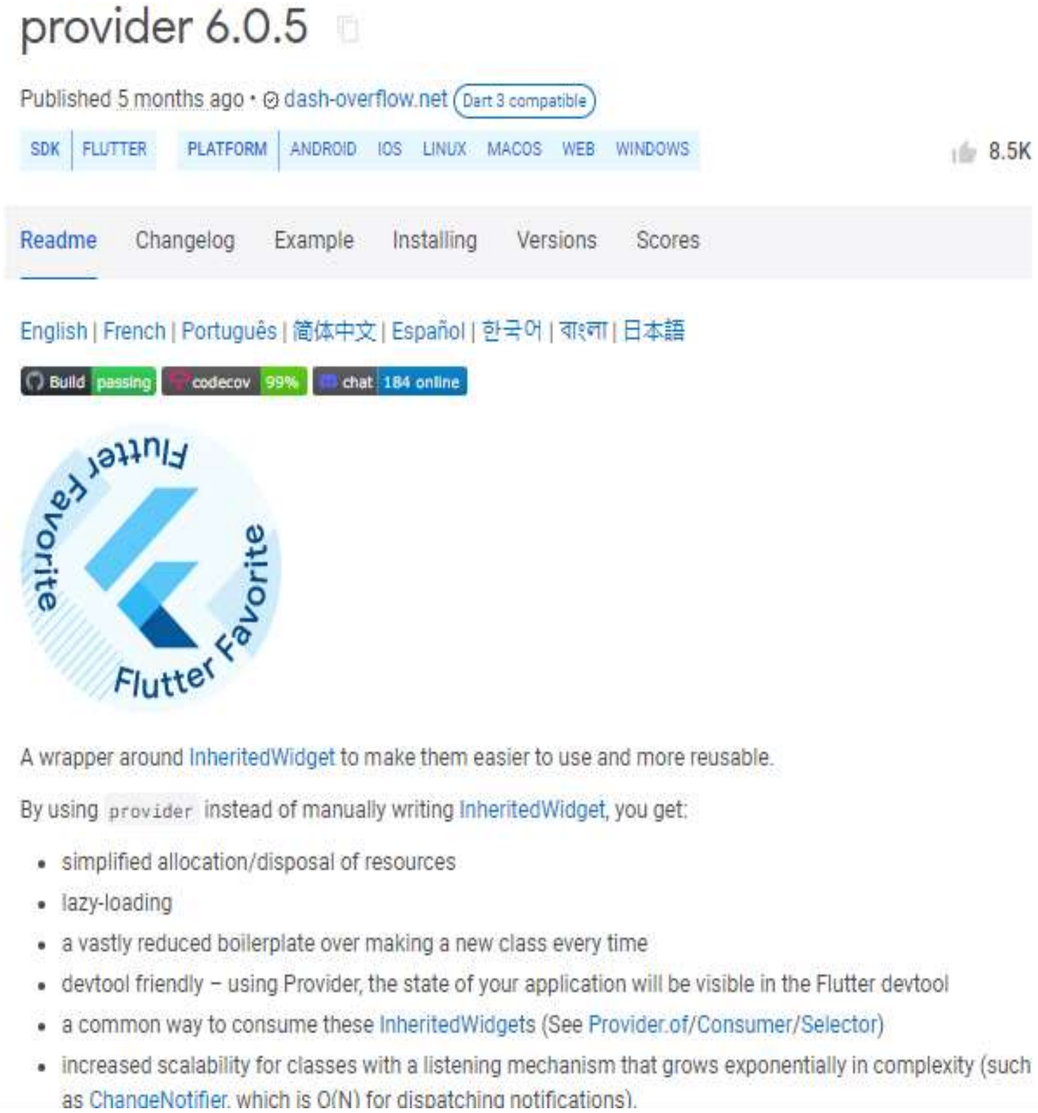
Provider

Provider adalah desain pola manajemen status yang digunakan untuk mengelola dan membagikan data ke berbagai bagian aplikasi.
Provider memungkinkan komponen yang terkait untuk berlangganan perubahan data dan memperbarui tampilan secara otomatis.

MOJADIPRO

Provider Package

Wrapper di sekitar InheritedWidget adalah "Provider," yang merupakan bagian dari paket Provider di Flutter. Provider dirancang untuk menyederhanakan pengelolaan dan penggunaan InheritedWidget, sehingga lebih mudah digunakan dan lebih dapat digunakan kembali.



MOJADIPRO

Menggunakan Provider Package

Langkah 1: Menambahkan dependensi **provider** ke file **pubspec.yaml** di proyek Flutter Anda:

```
dependencies:
  flutter:
    sdk: flutter
  provider: ^6.0.1
```

Lalu, jalankan **flutter pub get** untuk mengunduh dan menginstal dependensi.

Langkah 2: Membuat Model yang akan digunakan untuk menyimpan state aplikasi. Misalnya, jika Anda ingin menyimpan data pengguna, buat file **user_model.dart** dan tambahkan kode berikut:

```
class UserModel {
  String name;
  int age;

  UserModel({required this.name, required this.age});
}
```

MOJADIPRO

Menggunakan Provider Package

Langkah 3: Membuat ChangeNotifier. Buat file **user_provider.dart** dan tambahkan kode berikut:

```
import 'package:flutter/foundation.dart';

class UserProvider extends ChangeNotifier {
  UserModel _user = UserModel(name: '', age: 0);

  UserModel get user => _user;

  void updateUser(UserModel newUser) {
    _user = newUser;
    notifyListeners();
  }
}
```

UserProvider adalah kelas yang meng-extend **ChangeNotifier**. Ini akan mengurus state dan memberi tahu listener saat state berubah. Di sini, kami menggunakan _**user** sebagai state yang akan diperbarui.

MOJADIPRO

Menggunakan Provider Package

Langkah 4: Menggunakan Provider di Widget apa pun di aplikasi Anda untuk mengakses dan memperbarui state. Misalnya, di dalam widget **MyHomePage**,

Di sini, kami menggunakan **Provider.of<UserProvider>(context)** untuk mengakses instance **UserProvider** yang diperlukan. Kami kemudian mengambil state **user** dari provider dan menggunakannya dalam widget. Ketika tombol floating action button ditekan, kita memperbarui state menggunakan **userProvider.updateUser(newUser)**.

MOJADIPRO

Menggunakan Provider Package

Anda dapat menggunakan kode berikut:

```
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';

class MyHomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final userProvider = Provider.of<UserProvider>(context);

    return Scaffold(
      appBar: AppBar(
        title: Text('State Management with Provider'),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text('Name: ${userProvider.user.name}'),
            Text('Age: ${userProvider.user.age}'),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          final newUser = UserModel(name: 'John', age: 25);
          userProvider.updateUser(newUser);
        },
        child: Icon(Icons.add),
      ),
    );
  }
}
```

MOJADIPRO

TERIMA KASIH